Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Profiling and annotation combined method for multimedia application specific MPSoC performance estimation*

Kai HUANG†1, Xiao-xu ZHANG[1], Si-wen XIU†‡2, Dan-dan ZHENG[1], Min YU[1], De MA[3], Kai HUANG[4], Gang CHEN[4], Xiao-lang YAN[1]

([1]*Institute of VLSI Design, Zhejiang University, Hangzhou 310027, China*)

([2]*College of Optical and Electronic Technology, China Jiliang University, Hangzhou 310018, China*)

([3]*Microelectronics CAD Center, MOE Key Lab of RF Circuits and Systems, Hangzhou Dianzi University, Hangzhou 310018, China*)

([4]*Department of Informatics VI, Technical University Munich, Garching 85748, Germany*)

†E-mail: huangk@vlsi.zju.edu.cn; xiusw@vlsi.zju.edu.cn

Received July 5, 2014; Revision accepted Oct. 22, 2014; Crosschecked Dec. 30, 2014

Abstract: Accurate and fast performance estimation is necessary to drive design space exploration and thus support important design decisions. Current techniques are either time consuming or not accurate enough. In this paper, we solve these problems by presenting a hybrid method for multimedia multiprocessor system-on-chip (MPSoC) performance estimation. A general coverage analysis tool GNU gcov is employed to profile the execution statistics during the native simulation. To tackle the complexity and keep the analysis and simulation manageable, the orthogonalization of communication and computation parts is adopted. The estimation result of the computation part is annotated to a transaction accurate model for further analysis, by which a gradual refinement of MPSoC performance estimation is supported. The implementation and its experimental results prove the feasibility and efficiency of the proposed method.

Key words: MPSoC, Gradual refinement, Native simulation, Performance estimation, Profiling, Annotation, Gcov
doi:10.1631/FITEE.1400239 **Document code:** A **CLC number:** TP36; TN47

1 Introduction

Performance estimation is becoming a very important and challenging task in heterogeneous multiprocessor system-on-chip (MPSoC) design (Posadas *et al.*, 2004). Accurate and fast performance estimation in an earlier stage is necessary to drive design space exploration, which would avoid costly design process iterations (Gerin *et al.*, 2009) and fit tight time-to-market and time window constraints. Since MPSoC is becoming more and more complex, several abstract models have been employed (Jerraya and Wolf, 2004). Performance estimation should take a different focus on different abstraction levels and be gradually refined. This idea is inspired by previous research (Keutzer *et al.*, 2000; Jerraya *et al.*, 2006; Huang *et al.*, 2009). Huang *et al.* (2009) introduced a gradual refinement flow using five different abstraction levels, which are, from high to low, Simulink combined algorithm and architecture model (CAAM), virtual architecture (VA), transaction accurate (TA), virtual prototype (VP), and field-programming gate array (FPGA) emulation. Fig. 1 shows an example of the refinements of

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 61100074), the National Science and Technology Major Project of China (No. 2012ZX01039-004), and the Fundamental Research Funds for the Central Universities, China

ORCID: Kai HUANG (first author), http://orcid.org/0000-0002-5034-7171; Si-wen XIU, http://orcid.org/0000-0003-0400-8037

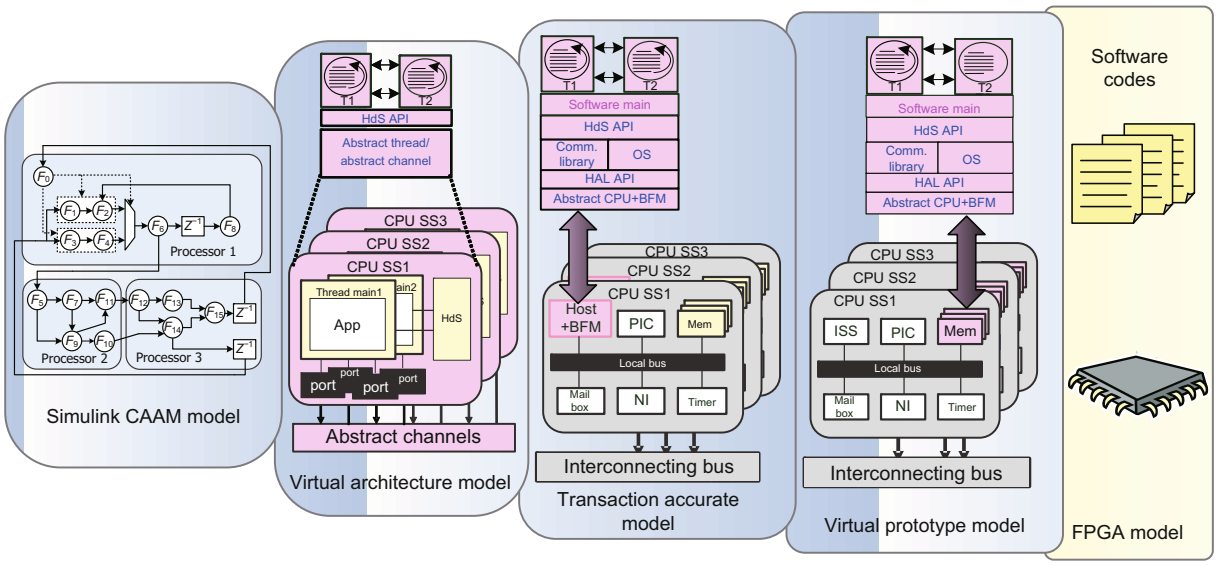

Fig. 1 An example of the refinements of the software model

MPSoC design flow with all these abstraction models (or levels). In this paper, CAAM is defined as a software and hardware combined model, built from an abstract clock synchronous model (ACSM) (Han *et al.*, 2006). Compared with data-driven synchronous data flow (SDF), Kahn process networks (KPN), and other functional models, ACSM is a synchronous model with capability to describe both data flow and control structure. To some extent, ACSM is close to homogeneous SDF (HSDF) with additional control tokens.

A critical issue for MPSoC performance estimation is to evaluate the expected performance early in the design process without actual hardware implementation (Yang *et al.*, 2010), and using abstraction level models leads to the results in seconds for cases in which cycle-accurate simulation takes tens of minutes or hours (Han *et al.*, 2009). The VP model is not suitable for performance estimation because of low simulation speed and low abstraction level, and neither is the FPGA model because of the low level and difficult implementation. As to the Simulink CAAM model, architecture mapping has not been decided, so it is too abstract to analyze. The VA and TA models are the most attractive for MPSoC performance estimation.

At the VA level, HW/SW partitioning and resource allocation are made explicit, and the allocation of threads/tasks to a subsystem is also fixed (Jerraya *et al.*, 2006), which makes VA the highest level that is suitable for performance estimation. The VA model is composed of some abstract CPU subsystems communicating with each other via abstract communication channels (Huang *et al.*, 2009) whose computation part is explicit while the communication part is not. The TA model goes further, detailing the local architecture of each subsystem in MPSoC and making the communication model explicit (Han *et al.*, 2009). The computation parts of the VA model and the TA model are almost the same, while the communication part of the VA model is represented by send_data/receive_data and that of the TA model by write_mem/read_mem because its memory subsystem and communication channels are made explicit.

Current MPSoC involves two different kinds: general-purpose multicore architecture like Cortex A9, and application-specific MPSoC like application-specific instruction-set processor (ASIP) based MPSoC or fine-grained processing element (PE) based system-on-chip. This study aims at multimedia application specific MPSoC performance estimation. The hardware platform used in this study employs simple processor architecture and explicit memory hierarchy. In such an architecture, communication overhead plays an important role, and this should be simulated explicitly in TA-level simulation. Computation load is less important and can be abstracted taking advantage of VA-level native simulation. Moreover, Keutzer *et al.* (2000) showed that orthogonalization of computation and communication is essential to master system design complexity,

which inspires us to separate the focuses of performance estimation at different abstraction levels. We thus propose a VA and TA combined strategy to profile computation load during VA simulation and annotate it to TA simulation, so as to achieve good trade-offs between estimation speed and accuracy.

This paper focuses on how to calculate the execution time with MPSoC abstraction models rapidly and accurately. We use a GNU compiler collection (GCC) profiling tool GNU gcov to collect the execution statistics of the given C code during simulation. GNU gcov is traditionally used to optimize the application code itself by rewriting the hot spots for more efficient execution on the target machine (GNU, 2013). Inspired by Karuri *et al.* (2005), here we use it in connection with a C compiler to generate statistics about the execution frequencies of code lines. To make the simulation fast, we employ native simulation where the applications are executed on a workstation or server with x86 CPU (Gerin *et al.*, 2009). Jerraya *et al.* (2006) showed that native simulation with the VA model can achieve 100 times the speed of the TA model with only a 15% accuracy loss. Thereby, we can estimate the performance with the MPSoC VA model at high speed, and the result of its computation part can be annotated to the TA model. Thus, only the communication part needs to be analyzed for a more accurate estimation at the TA level. We call the annotation flow 'gradual refinement of performance estimation'. Our contributions are summarized as follows:

1. We propose a profiling and annotation combined flow for multimedia MPSoC performance estimation from the VA level to the TA level.

2. We propose a profiling based method for VA-level native simulation. GNU gcov is employed to profile the execution statistics of the given C code during native simulation, and runtime performance analysis is supported, which enables accurate and fast VA-level performance estimation.

3. We propose an annotation based simulation method for TA-level transaction accurate simulation. Only the communication latency is refined, which enables more efficient TA-level performance estimation.

2 Related work

In the past few years, different techniques have been proposed for heterogeneous MPSoC performance estimation, and most of them fall into three categories: execution-driven simulation, analytic method, and a hybrid of both.

Execution-driven simulation is usually used for design verification. For the hardware part, simulation performance depends on the abstraction level, and for the software part, instruction-set-simulator (ISS) is the simplest technique (Fummi *et al.*, 2004; Benini *et al.*, 2005; Filho *et al.*, 2008). Even though ISS simulation benefits from high accuracy in contrast to other approaches, it suffers from such a low simulation speed, costly setup, and lack of flexibility that it can be employed only in the final stage of the development when the design space has been significantly narrowed down.

To speed up the estimation process, some analytic methods are introduced which remove the need for the cumbersome simulation. They usually consider all possible paths in the control flow graph (CFG), and are widely used to calculate the worst-case execution time (WCET) (Wilhelm *et al.*, 2008) for real-time systems. SymTA/S (Richter *et al.*, 2003; Henia *et al.*, 2005) is a formal representative compositional approach, which uses schedulability analysis techniques. Modular performance analysis (Wandeler *et al.*, 2006; Huang *et al.*, 2012) is another compositional approach, which relies on real-time calculus. Madl *et al.* (2007) and Yang *et al.* (2010) both used an exhaustive enumeration approach to enumerate all possible event execution paths and then verified them. These analytic approaches compute the delays instead of simulating them, and trade accuracy for estimation cost and speed, which are more valuable in the early design phases. However, one problem is their limited ability to carry out the real workload scenarios. The other drawback is that they rely only on worst-case analysis, the results of which are too conservative.

Recently, some hybrid methods have been proposed to resolve performance issues by combining advantages of simulation-based and analytic approaches. Oyamada *et al.* (2007; 2008) employed a cycle-accurate simulator to extract the execution results of a given application, which are used as the input for neural networks (NNs) for training and utilization. This kind of NN method can easily adapt to the non-linear behavior such as pipelines, branch prediction, and caches, and the estimation speed is fairly fast. However, it needs a training

process that is costly and considerably long, and any architecture modification would require a new one. Piscitelli and Pimentel (2012) interleaved analytic performance estimations with simulations, providing trade-off between speed and accuracy, but their focus is only on design space exploration on system-level mapping. To speed up the simulation part, native simulation approaches have been proposed (Gao *et al.*, 2008; Gerin *et al.*, 2008; Shen *et al.*, 2012), where software runs on the host machine natively, whose execution time is usually calculated through delay-annotation. Schnerr *et al.* (2008) applied back-annotation of WCET/BCET values. Unfortunately, the WCET/BCET values are not accurate enough for the real workload as mentioned above. Kirchsteiger *et al.* (2008) first obtained the target assembly codes from the target cross-compiler, and then analyzed them and generated delay information for each C-code statement according to the target processor datasheet. After that, the delay information is annotated to the original C code, which is finally executed natively to calculate the real delay. The delay-annotation techniques could replace the ISS with a faster and more accurate delay-annotated software model. However, software codes have to be largely modified with delay insertion and the pipeline model is coarse.

Our work aims at combining the advantages of analytical and simulation techniques, and providing high speed and accuracy for efficient performance estimation. The key idea is to profile the execution information and calculate the computation load during VA level native simulation, and the results can be annotated to the TA model, by which the performance estimation is able to be gradually refined. Compared with previous techniques, ours inserts few extra codes into the original program codes, and can provide runtime analysis.

3 Annotation based method at the transaction accurate level

To gradually refine the MPSoC performance estimation, we use an annotation method among different abstraction levels of MPSoC.

The VA model consists of some abstract CPU subsystems communicating with each other via abstract channels, while interconnect components, memories, and memory mapping are not decided, which makes the processors take the responsibility for all the data transfers of send_data/receive_data functions. The TA model details the local architecture of each subsystem in MPSoC and makes the communication protocol explicit. CPU subsystems communicate with each other via memory subsystems and cycle-accurate communication channels. Each CPU subsystem of the TA model is composed of an abstract CPU that provides bus functional model (BFM) functions and translates them to SystemC signal-level transactions, local buses and memories, a mailbox for data synchronization among the CPU subsystems, an interrupt controller (INTC) to manage external interrupts generated from the mailbox, local memories, a network interface (NI) to convert global addresses to local addresses to allow remote access from other CPU subsystems, and possibly a direct memory access (DMA) to take responsibility for data transfers between the local memories and the global memory, which reduces the burden on the processor. Thereby, the communication parts of both intra- and inter-CPU-subsystem are refined and become closer to the real chip. Performance estimation should then be made again to refine the result. Since the architecture of each CPU subsystem has already been decided at the VA level, the computation cost of VA is as accurate as that of TA, which leads us to annotate the VA performance results of the computation part back to TA. Details about TA-level simulation and how communication works can be found in the literature (Jerraya *et al.*, 2006; Han *et al.*, 2009; Huang *et al.*, 2009).

Fig. 2 illustrates an example of the annotation flow from VA to TA. When prf_end (a profiling API function, explained in Section 4) is called, in the analyzing process the performance results are calculated and the corresponding codes generated for TA level performance estimation. If the message type of this prf_end is assigned 0 (computation), the computation part is commented out, being replaced by function prf_annotation, which is used only for TA level performance estimation, whose parameter is the cycle count cost of the corresponding computation part.

Fig. 3 illustrates a TA platform using our profiling and annotation based performance estimation method, whose essence is to use computation cost obtained from VA and estimate the communication cost using cycle approximate simulation. For hardware,

except for CPUs, all other components are implemented with cycle-accurate SystemC models, using a bus function model (BFM) and Linux shared memory for communication. For software, it is not necessary to run the computation part, whose cost is annotated and directly profiled. However, the communication part needs to be executed and profiled, because it has some peripheral configuration operations, such as DMA register configuration. Since our purpose is to estimate the performance instead of running a verification, the exact read/write data are not important and are substituted by fake ones.

As shown in Fig. 3, the execution time (e.g., time1 and time4) of the computation part is obtained in advance from the VA level, and the communication configuration time (e.g., time2 and time5) is obtained by runtime profiling and analyzing. By adding them, the execution time between two read/write operations (e.g., time3 obtained by time1 plus time2, and time6 obtained by time4 plus time5) is obtained and sent to the BFM in hardware simulation of the TA model through the Linux shared memory. By this means, the native simulation of application software cooperates with the hardware simulation of a communication model, which combines computation cost with communication latency for the overall performance estimation of the system.

Thanks to our annotation technique, the software codes generated during the performance estimation at the VA level can be reused at the TA level and only the communication part needs to be refined or rewritten, which makes it faster and more efficient for performance estimation.

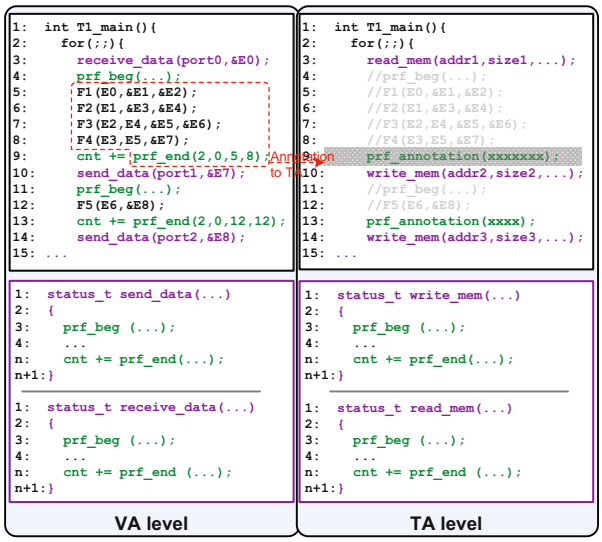

Fig. 2 Annotation flow from the virtual architecture (VA) level to the transaction accurate (TA) level

4 Profiling based method at the virtual architecture level

4.1 Workflow

Profiling and simulation are a pair of inherent techniques for acquiring performance information of an application (Patel and Rajawat, 2011). We apply native simulation to achieve fast execution speed, and use a mature profiling tool, GNU gcov, to obtain accurate information. Based on these two novel techniques, we can analyze the pipeline in a fine-grained way at runtime.

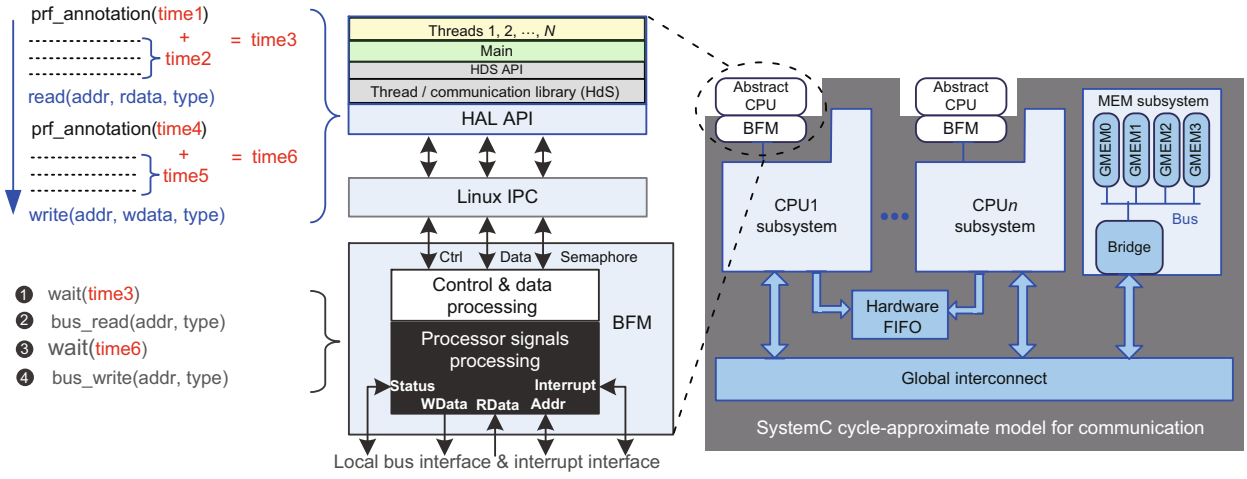

Fig. 3 Profiling and annotation based transaction accurate platform

Fig. 4 presents the four main steps of our estimation method: (1) profiling API insertion, (2) double compilation, (3) runtime performance analysis, and (4) result annotation.

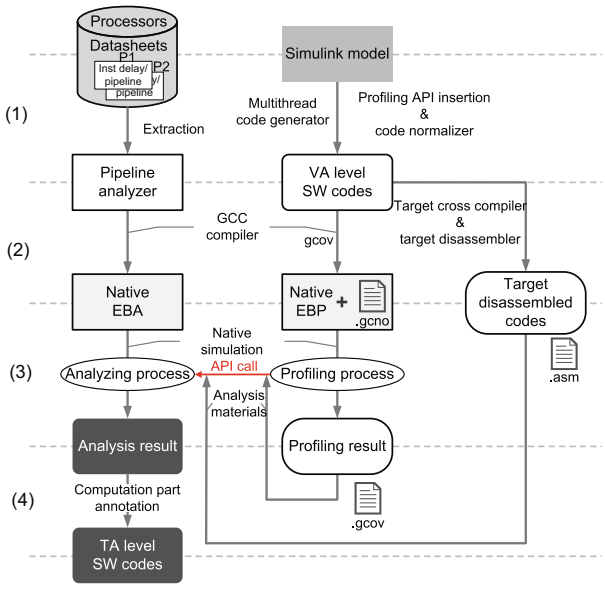

Fig. 4 Workflow of the performance estimation method in this study

In the API insertion phase, some profiling API functions are inserted into the multithread codes generated by Simulink at the Simulink CAAM level. Thanks to the API insertion (Section 4.3), the profiling and analysis can be done together at runtime, which makes our method more efficient.

In the double compilation phase, the multithread codes generated in the API insertion phase are compiled into an executable binary for profiling (EBP) by using GCC coverage parameters (gcov, refer to Section 4.2). Some extra files of gcov are generated. In addition, an executable binary for analysis (EBA) is generated from the pipeline analyzer (Section 4.4) by the GCC compiler. As the full name implies, EBP is responsible for native simulation while EBA for performance analysis, the generation of which is called 'compilation for runtime analysis' in this study. Meanwhile, a target executable binary is generated and then disassembled to the target disassembled codes using the tool chain of the target CPU architecture, such as the cross compiler and disassembler, the process of which is called 'compilation for analysis materials'.

4.2 Gcov

In the runtime performance analysis and annotation phases, both of the generated binary executables are run on the host machine. The performance results are generated, recorded, updated, and annotated according to the calling of the API functions. Section 4.3 shows more details.

To ease the native simulation, we adopt a general profiling tool GNU gcov. It belongs to source code level profiling which is the most widespread, and is suitable for our Simulink based design flow because the multithread codes to be analyzed are usually in form of C or C++. By using gcov, we can find some basic performance statistics, such as how often each line of code is executed, and which lines of code are actually executed.

Two files generated by gcov play an important part in our estimation process, named .gcno and .gcov files. As shown in Fig. 5, the .gcno notes file (here tmp.gcno) is generated when the source file is compiled by a GCC compiler with gcov. It contains information to reconstruct the basic block graphs and assign source line numbers to blocks. During the native simulation, gcov creates a log file named sourcefile.gcov (here foo.c.gcov) which indicates how many times each line of a source file sourcefile.c (here foo.c) has been executed (GNU, 2013). The coverage log file (.gcov) makes it possible to estimate the execution time of a given application. However, to analyze the performance of the application on the target platform, we employ the target cross the compiler and disassembler to generate the disassembled code where each C-code statement is mapped to its corresponding assembly instructions of the target architecture while keeping the source line number. Given the coverage log and the disassembled code, we can analyze the running status of the target pipeline according to the datasheet of the target processor, and estimate how much computing time each section of the codes uses. The pipeline analyzer will be explained in Section 4.4.

However, there are two restrictions on the gcov profiling based analysis method. One is that GCC compiler optimization should not be used if high accuracy performance estimation is desired. Once GCC compiler optimization is used, the mapping between C-code statements and their assembly codes is probably disturbed, which makes the pipeline

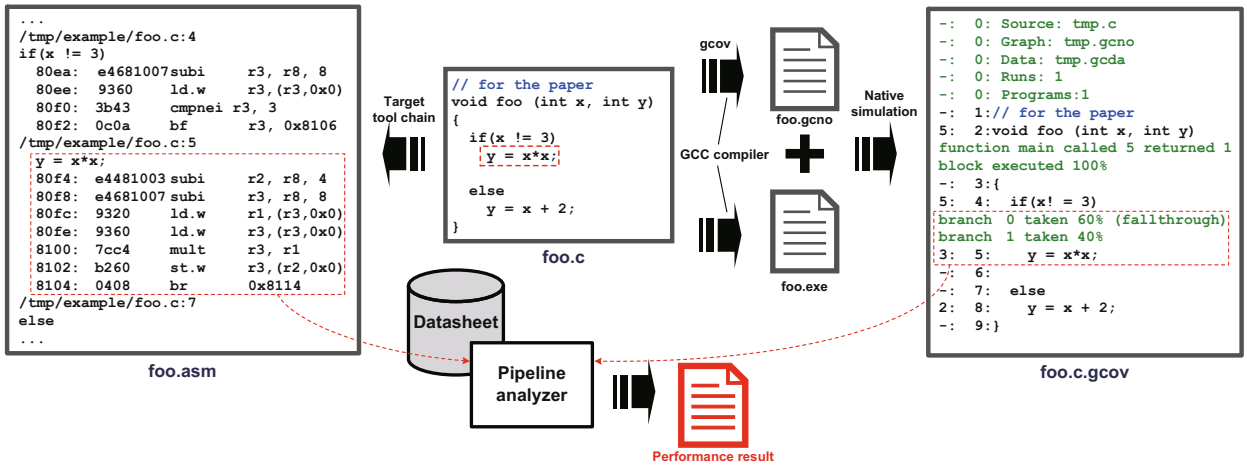

Fig. 5 GNU gcov based profiling and analyzing method

analyzer not function well. With our experience from the experiments, the estimation error caused by GCC optimization is about 3%–54% according to the optimization level. If we use the optimization level (-O1, -O2, -Os) less than (-O3), the estimation error is not more than 12%, which is suitable for architecture exploration with horizontal comparison between different architectures in most cases. We do not suggest using a high optimization level (-O3) in our work since then a too inaccurate result can be avoided. The other restriction is that some programming rules must be followed. Since gcov accumulates statistics by line, it works best with a coding style that places only one statement on each line. However, rules like this are often violated. To solve the problem, we adopt a code normalizer to modify the man-made codes.

There are four kinds of codes that are modified by the code normalizer:

1. Multiple assignment statements in one line. As shown in Fig. 6a, they are distributed to multiple lines, guaranteeing that there is one assignment statement in one line at the most, which is a basic rule of the code normalizer.

2. Assignment statement with ternary conditional operator '?:'. As shown in Fig. 6b, the statement is reconstructed in the form of 'if/else' conditional assignment.

3. Conditional branches in the same source code line. As shown in Fig. 6c, they are distributed.

4. Control structure and assignment statement in one line. As shown in Fig. 6d, they are also distributed.

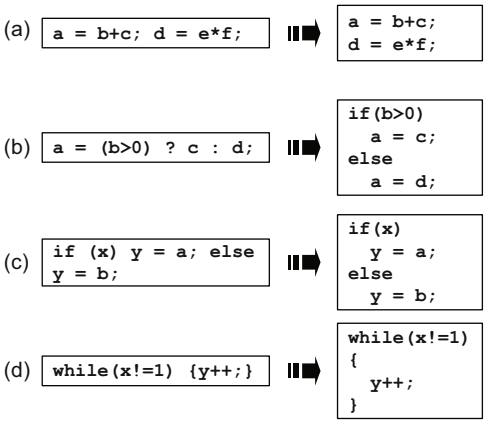

Fig. 6 Cases of the code normalizer: (a) two uncorrelated assignment statements in a single line distribute into two lines; (b) ternary-statement translates into standard conditional statements; (c) condition statements in a single line distribute into several lines; (d) control structure and assignment statement distribute into different lines

4.3 Profiling API functions

4.3.1 Definition

The main purpose of profiling API functions is to manage performance estimation tasks required for native simulation.

Function anl_fork is called at the beginning of the native simulation, when some environment variables need to be initialized or configured. It is used to fork a new child process where the EBP runs from the parent process in which the EBA runs. We call the parent process the profiling process and the child process the analyzing process. The analyzing process then keeps waiting for a profiling API call from

a profiling process as soon as the environment is set up.

Functions prf_beg and prf_end are used as starting and ending points for the profiling task, respectively. The function prf_beg is able to clean the previous profiling states and prepare a new environment for the subsequent profiling task, and the function prf_end is used to send a message to the analyzing process to activate it. prf_end is the most important API function, because it is the only one that enables interaction between the profiling process and the analyzing process. As shown in Fig. 7, function prf_end carries the following information for performance analysis: CPU ID is used to inform the analyzing process which processor architecture shall be chosen. Message type is employed to indicate whether the codes to be analyzed belong to the computation or communication part. The number of the start line and that of the end line are also given to specify the portion of the codes to be analyzed. Last but not the least, prf_end is also responsible for the annotation.

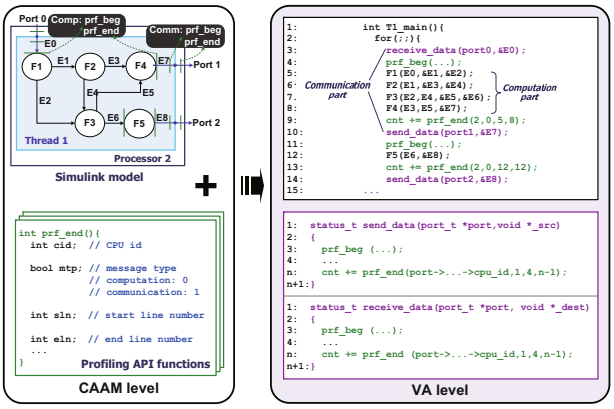

Fig. 7 Profiling API functions insertion

Function anl_join is used to complete the analyzing process before terminating the whole simulation.

4.3.2 Insertion

Fig. 8 shows how to insert the profiling API functions into the generated software code stack at the VA level and TA level, respectively. At the VA level, API function calls for the performance of computation are inserted into application thread codes directly, while those calls for the performance of communication are inserted into the abstract communication library, like send_data and receive_data. At the TA level, the same application thread codes are reused while some main hardware dependent software (HdS) codes are refined for the target platform. Thereby, API function calls for the performance of communication are inserted into the real communication library, and some other API function calls for the delay cost of synchronization are also inserted into some event functions in this library, like sent_event and receive_event. Moreover, some special API function calls are inserted into the multithread library to calculate OS performance and total idle time of one processor when no thread is running.

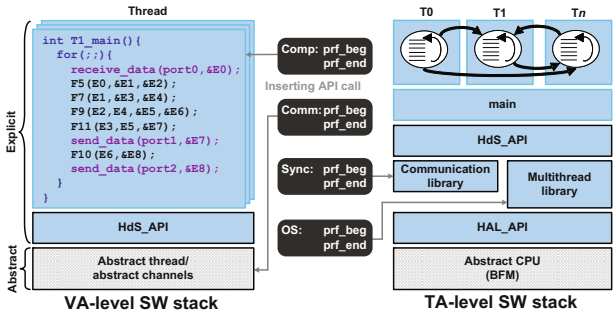

Fig. 8 Profiling API functions insertion of software code stack

Fig. 7 shows how to insert functions prf_beg and prf_end from the Simulink CAAM level to the VA level in more detail. From the viewpoint of the Simulink CAAM level, prf_beg and prf_end are inserted into the starting and ending points of one port in the Simulink thread model, respectively. From the viewpoint of the VA model, prf_beg/prf_end has been inserted at the top/bottom of each communication or computation block. We take as an example the computation block that consists of F1, F2, F3, and F4. At the VA level, a prf_beg has been inserted at line 4 right on top of F1 and a prf_end has been inserted at line 9 right after F4. The CPU ID is assigned 2 according to the Simulink model, the message type is assigned 0 which indicates it is a computation block, and the portion of the codes to be analyzed is specified from line 5 to line 8. For the communication block, the API insertion makes no difference except for the CPU ID that will be derived from the Simulink structure.

4.3.3 Profiling and analyzing

During the native simulation, the profiling process and the analyzing process run on the host

machine. Fig. 9 shows the runtime analysis phase controlled by the API function call.

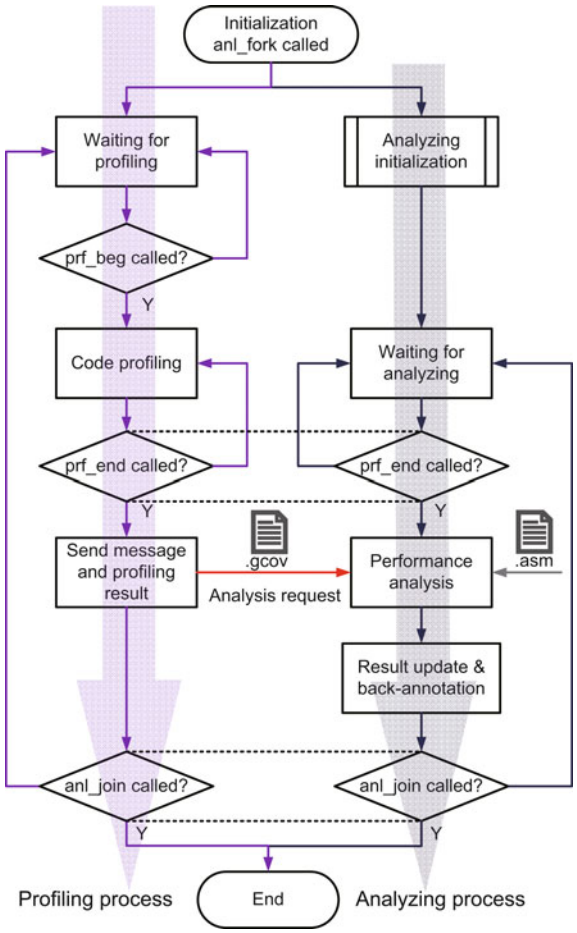

Fig. 9 Runtime analysis phase

After the initialization, the profiling process begins to search down the codes to be analyzed. If a prf_beg is called, some previous profiling data are cleaned, and the subsequent codes are profiled until the corresponding prf_end is called. Obviously, prf_beg and prf_end always come in pairs. When prf_end is called, the profiling process sends messages and the profiling result (files suffixed by .gcov) to the analyzing process for further analysis. The profiling process keeps snooping and handling the function calling for prf_beg and prf_end until anl_join is called, which terminates the whole simulation.

As for the analyzing process, it is forked when anl_fork is called. After initialization, it keeps waiting for function calling for prf_end. When prf_end is called, the analyzing process invokes the pipeline analyzer to estimate the performance of the codes to be analyzed, with the profiling result provided by the profiling process and the disassembled codes provided by the compilation for analysis materials. The analyzing process keeps waiting for and handling the analysis requests until anl_join is called.

4.4 Pipeline analyzer

As mentioned above, the pipeline analyzer is used to calculate the cycle cost of each assembly instruction during the analyzing process. In this subsection, we show the functions and the workflow of the pipeline analyzer in detail. As shown in Fig. 10, the pipeline analyzer consists of three parts: coverage parser, instruction-set lookup-table (ISLT), and pipeline parser. The coverage parser is employed to extract the inputs, the ISLT provides the database for analyzing, and the pipeline parser is employed to generate the outputs.

4.4.1 Coverage parser

The coverage parser is designed to parse the information carried by the API function prf_end, such as the codes to be analyzed and the corresponding processor model that is executing them, and to extract the execution statistics of each line of the codes from the .gcov files. Meanwhile, the corresponding assembly instructions are extracted from the disassembled codes generated by the target tool chain of the given processor architecture. If there is no function call in the current line, the pipeline parser is called to calculate the performance result. If the current statement calls another function, the coverage parser also parses the .gcov file of this function for the pipeline parser.

4.4.2 Instruction-set lookup-table

To calculate the accurate cycle cost of each assembly instruction, the ISLT is employed. This is extracted from the datasheets of all the processor architectures provided in the design. Three primary influences are taken into consideration: instruction, pipeline, and memory latency. For instance, Table 1 takes ISLT for the CK803 processor (without I/D cache). CK803 (C-SKY Microsystems, 2013) is a low-power, high-performance, and three-pipeline-stage processor, and some frequently used instructions in the C-SKY V2 instruction set that is used by CK803 are listed. For each instruction, its execution

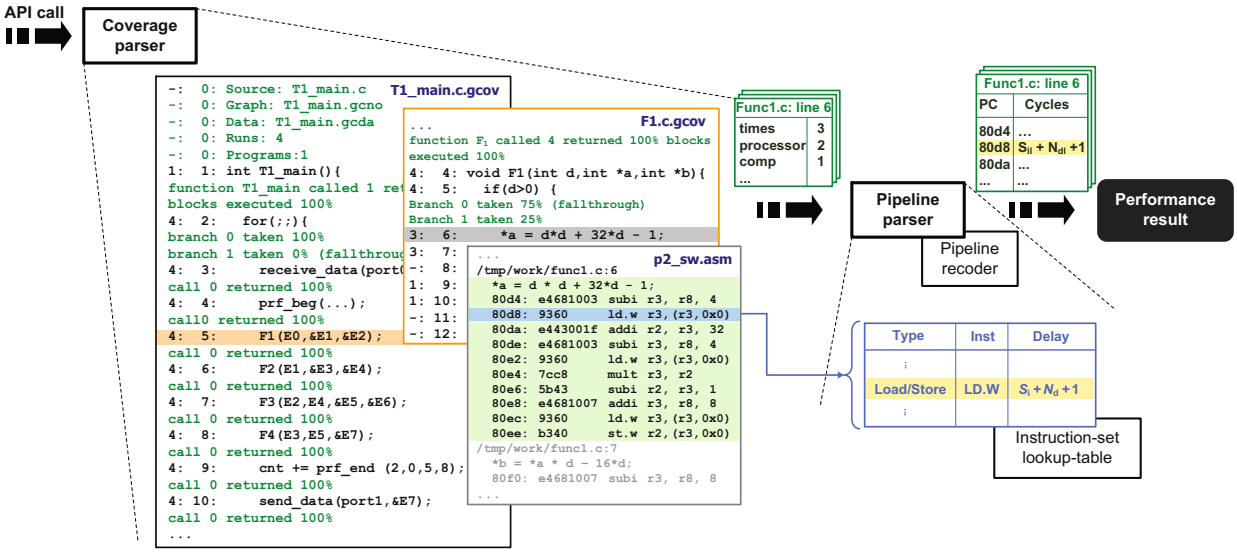

Fig. 10 Workflow of the pipeline analyzer

time consists mainly of instruction fetching, memory accessing, and executing. For some coprocessor instructions, additional time may be added.

Table 1 Instruction-set lookup-table of the CK803 processor

Type	Instruction	Delay
Load/Store	LD.W	$S_\mathrm{i}+N_\mathrm{d}+1$
	ST.W	$N_\mathrm{i}+N_\mathrm{d}$
	LDM	$S_\mathrm{i}+S_\mathrm{d}+(n-1)S_\mathrm{d}+1$
	STM	$N_\mathrm{i}+N_\mathrm{d}+(n-1)S_\mathrm{d}$
Data processing	ALU	S_i
	MULT	$S_\mathrm{i}+4$
	MULU	$S_\mathrm{i}+5$
Coprocessor INST	MFCR	$S_\mathrm{i}+C$
	MTCR	$S_\mathrm{i}+C+1$

The delay of each instruction consists mainly of three parts, which are instruction prefetch time, data memory access time, and instruction execution time. We define N as the number of cycles for a non-sequential data access of a single memory operation, and S the number of cycles for a sequential data access of continuous memory operations. The value of N may be different from that of S for special bus and memory access. For example, DRAM may take more cycles to access an address not related to the previous address. For further refinement at the TA level, S_i and N_i are defined as the cycle costs of the instruction memory access, while S_d and N_d are defined as those of the data memory access. What is more, the cycles of multiple-load/store instruction, such as LDM or STM of the CK803 processor, involve the cost of one non-sequential cycle for the first data access and $n-1$ (n is the total load/store times in this instruction) sequential cycles for the subsequent data access.

As for the instruction execution time, some instructions may need multi-cycle execution time. For the multiply instruction (MULT), the execution cycle depends on the input data, and it requires four cycles to complete in the worst case. Parameter C is used to represent the cycle cost of the coprocessor instructions.

4.4.3 Pipeline parser

Given the information extracted from the coverage parser, the pipeline parser calculates the cycle cost of each assembly instruction by searching the ISLT and referring to a pipeline recorder, which can record the status of the pipeline and imitate its behavior.

Provided by the ISLT, the cycle cost of an assembly instruction can be computed according to formula $T_\mathrm{inst}=T_\mathrm{delay}+T_\mathrm{hazard}$. Here T_hazard represents the cycle cost of pipeline stall. The pipeline recorder is employed to recognize a pipeline stall such as register hazard.

We take the LD.W instruction in Fig. 10 as an example. The cycle cost is $T_\mathrm{LD.W}=S_\mathrm{i}+N_\mathrm{d}+1$ (no hazard). However, it is not the final result because

whether this access is to global memory or local memory is not established. We assume that a processor fetches most of the data from its local memory, and that it uses global memory only for communication with the other processors, which is common in heterogeneous MPSoC design. Even though the delays of memory accesses are not decided, we can replace them with parameters. In our VA model, for the computation part, every memory access is to the local memory, and the delay is defined as S_{il}, N_{il}, S_{dl}, or N_{dl}, which adds an 'l' to the subscript of the representation of the memory access delay. As for the communication part, the send_data and receive_data functions are both executed by the CPU that works like a DMA at the VA level. The instruction memory accesses are also to the local memory, while the data memory accesses depend. In the send_data function, the read_data memory instructions access only the local memory, whose delay is defined as S_{dl} or N_{dl}, while the write data memory instructions access only the global memory, whose delay is defined as N_{dg} or S_{dg}, which adds a 'g' to the subscript of the representation of the memory access delay. As for the receive_data function, all the data memory accesses are to the global memory. Therefore, the cycle cost of the LD.W in the computation part is modified to $S_{\mathrm{il}}+N_{\mathrm{dl}}+1$, while that in the receive_data function will be modified to $S_{\mathrm{il}}+N_{\mathrm{dg}}+1$.

Equipped with the pipeline parser, the delay of each assembly instruction can be calculated in a fine-grained way. There are some parameters in the result, such as N_{dg} and S_{il}, which represent some factors that cannot be decided at the VA level. However, the result in this form is especially helpful for design exploration.

5 Environment of use

We have implemented the proposed techniques in the Simulink-based MPSoC design platform (Huang *et al.*, 2007; 2009; Han *et al.*, 2009), which enables systematic and automated MPSoC design from a high-level algorithm specification using the Simulink environment and SystemC language. As shown in Fig. 11, this platform works as a typical Y-chart system-level design flow, which consists of application and hardware mapping, performance estimation, and performance result feedback (Kienhuis *et al.*, 1997; Keutzer *et al.*, 2000). The workflow of this platform is composed of four stages: Simulink modeling, computation-accurate performance estimation, cycle-approximate performance estimation, and cycle-accurate performance evaluation.

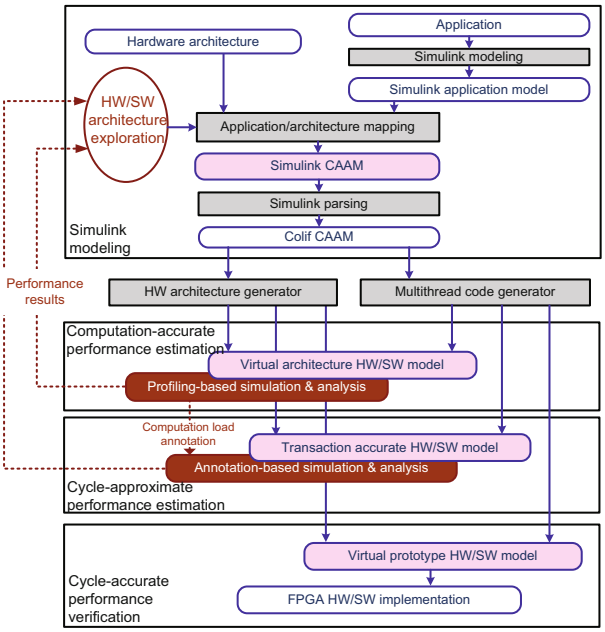

Fig. 11 Simulink-based MPSoC HW/SW co-design framework

In the stage of Simulink modeling, there are two inputs: hardware architecture and application model. After application and architecture mapping, we can obtain a CAAM to specify abstract hardware and software architecture. The Simulink parser traverses an input Simulink CAAM and generates an intermediate representation in Colif (Cesário *et al.*, 2001) for easy data manipulation. The Simulink parser also resolves the implicit types of Simulink links with type analysis, and its results are used in generating thread codes and implementing communication channels. Based on Colif CAAM, the hardware architecture generator generates multiprocessor hardware architecture models at three different abstraction levels, which are VA, TA, and VP. On the other hand, the multithread code generator produces embedded software stacks executing on the generated multiprocessor architecture models at the three different abstraction levels.

The next stage is computation-accurate performance estimation for early large-scale coarse-grained architecture exploration aiming at computation refinement. The profiling-based technique is used

in hardware and software co-simulation on the VA model. Taking advantage of the simulation and analysis combined method, we can obtain two kinds of result: a system performance result and a computation load of different functions. The performance results of the computation-accurate system can be fed back to help hardware/software architecture exploration obtain a better mapping model in the Simulink modeling stage.

The annotation-based technique is used in hardware and software co-simulation on the TA model. The function-level computation load results are annotated into the simulation with the communication model. The output of this performance estimation is cycle-approximate for small-scale fine-grained architecture exploration aiming at communication refinement.

The stage of cycle-accurate performance verification takes advantage of low-level VP model simulation and FPGA emulation to obtain cycle-accurate performance evaluation to check if the final performance result meets the design constraints.

6 Experiments

6.1 Experimental settings

In the experiments, we have adopted a flexible MPSoC software and hardware platform with good scalability and strong configurability. As shown in Fig. 12, the hardware platform consists of CPU subsystems, a memory subsystem, and an interconnection subsystem. Each CPU subsystem uses a 32-bit local bus to connect one processor with other local components and a memory service access point (MSAP) (Han *et al.*, 2004) to connect an external 64-bit AXI (ARM, 2003) bus-based distributed memory server (DMS) interconnecting network. The memory subsystem uses a 64-bit local bus to connect an on-chip global SRAM (GMEM) and an off-chip 16-bit DDR3 DRAM. The DMS acts as a server that provides the communication/synchronization services to the clients, that is, subsystems in an MPSoC. Each MSAP delivers data transfer requests issued by its corresponding subsystem to another MSAP via the control network. It also exchanges synchronization information, which indicates the completion of request handling, with other MSAPs via the control network. This platform has good scalability, capable of accommodating eight CPU subsystems. During architecture exploration, we can repartition the application with different thread numbers and also assign the tasks to different processors taking advantage of a load-balance strategy to achieve better performance. Global FIFO (GFIFO) mechanism is used for inter-processor communication and those buffers for thread communication. The platform also provides configurable HWFIFO for fast data synchronization from a real-time requirement. The software platform consists of thread codes, CPU main codes, and the HdS library on target CPU.

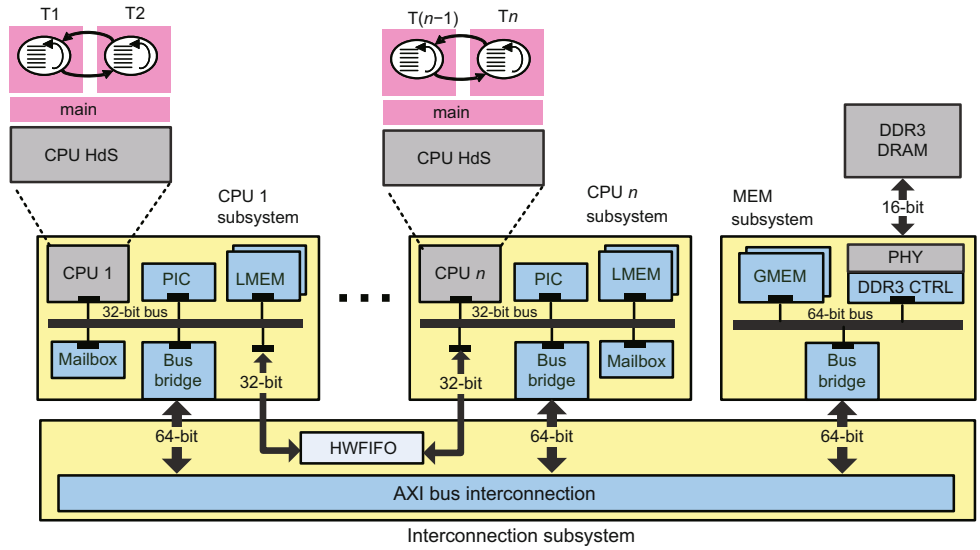

Fig. 12 MPSoC software and hardware platform template

We apply the Simulink-based MPSoC design platform using the proposed techniques in different MPSoC architectures with a 10-frame QVGA Motion-JPEG decoder and a 10-frame CIF MPEG2 video decoder. The Simulink functional model of Motion-JPEG decoder consists of 7 S-functions, 7 delays, 26 data links, and 4 if-action-subsystems (IASs). The Simulink functional model of the MPEG2 video decoder is more complicated with 49 S-functions, 18 delays, 186 data links, 28 IASs, 4 for-iteration subsystems (FISs), and 72 pre-defined Simulink blocks. Using the Simulink-based design platform, Simulink functional models are partitioned into a different number of threads and mapped into a 2–8-processor MPSoC hardware platform. The number of threads is scalable using the coarse or fine-granularity partition strategy on the Simulink CAAM. Fig. 13 shows an example of a 7-thread M-JPEG model mapped on a 4-processor MPSoC hardware platform.

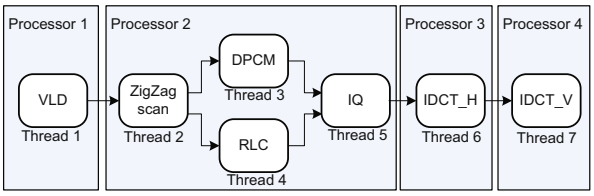

Fig. 13 Example of M-JPEG Simulink CAAM with mapping information

To efficiently evaluate the proposed method and its implementation, the architecture exploration tool in the Simulink-based MPSoC design platform uses the performance results fed back from different sources: VA, TA, and VP. As shown in Table 2, according to different sources for early or late stage in architecture exploration, we classify the estimation scheme into four different sets: TA, VP, TA+VP, and VA+TA. The first three schemes have been used for performance estimation and architecture exploration in previous work on the Simulink-based design platform (Han *et al.*, 2009; Huang *et al.*, 2009). The last scheme takes advantage of the profiling- and annotation-based techniques in the Simulink-based MPSoC design platform as described in Sections 3 and 4. In the experiments, given two decoder applications, these four schemes are used to estimate performance and explore architecture in the Simulink-based MPSoC design platform.

Table 2 Estimation scheme at different architecture exploration stages

Estimation scheme	Scheme	
	Early large-scale coarse-grained	Late small-scale fine-grained
TA	TA	TA
VA	VP	VP
TA+VP	TA	VP
VA+TA	VA+Profiling-based technique	TA+Annotation-based technique

6.2 Experimental results

We compare the experimental results of two key design efficiency factors, speed and accuracy, to show the feasibility and effectiveness of the proposed techniques.

6.2.1 Estimation speed

We use the total time of running the simulation and analysis on the host machine as the estimation time to evaluate the design speed. The host machine used in this experiment is four six-core Intel Xeon X5690 CPUs running at 3.47 GHz. As shown in Fig. 14a, given 4-processor MPSoC architecture, the average estimation time for 10-frame M-JPEG decoding with different estimation schemes, shows extremely different speeds. The most time-consuming scheme is VP, 2656.0 s, almost 177 times slower than the VA+TA scheme. After using the TA model for performance estimation in the early exploration stage, the average estimation time is very much reduced to 752.8 s because TA model simulation is much faster than VP-level simulation. If we use TA model simulation for both early coarse-grained exploration and later fine-grained exploration stages, the time consumed with the TA scheme is further cut to 103.6 s. Taking advantage of the proposed techniques, VA model simulation and analysis is used for the early exploration stage and TA model simulation speed is improved based on the annotation technique. The average estimation time of the VA+TA scheme is reduced to 15.2 s. In another experiment of MPEG2, the experimental results show a similar reduction ratio comparing the VA+TA scheme with the other three schemes. Fig. 14b shows the comparison of estimation time given different processor numbers from 2 to 8 in M-JPEG and MPEG2 decoder applications. The estimation time of the VA+TA scheme is changed less with the increasing

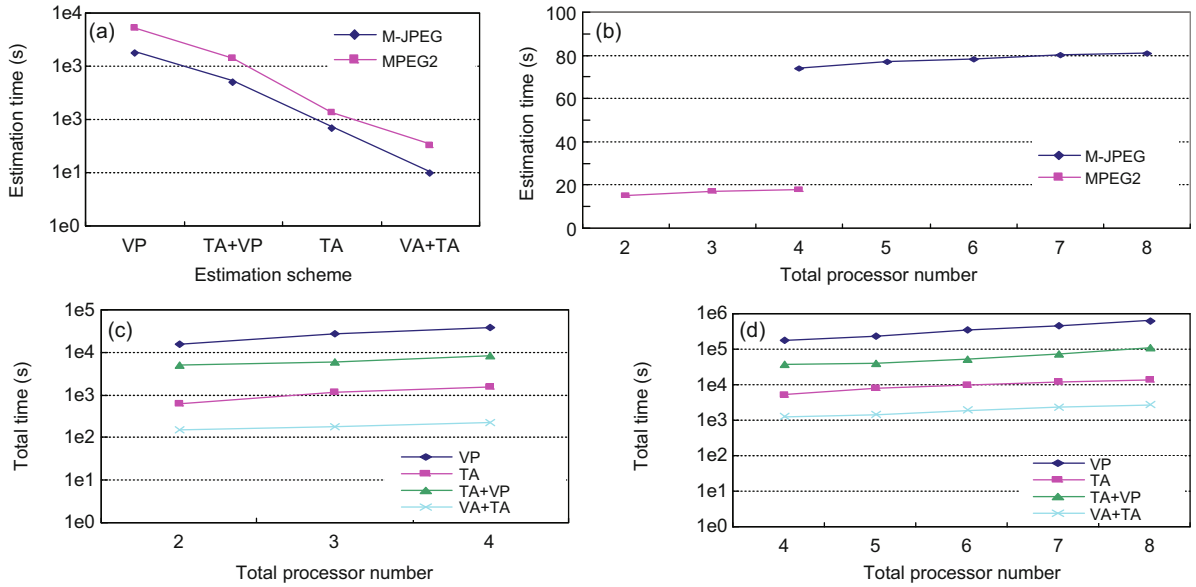

Fig. 14 Experimental results on estimation time: (a) estimation time of four different schemes for 10-frame M-JPEG and MPEG2 decoding; (b) estimation time of the VA+TA scheme given different numbers of processors in M-JPEG and MPEG2 decoder applications; (c) total time for four different schemes for MPSoC architecture exploration given 2, 3, and 4 processors in the M-JPEG decoder application; (d) total time for four different schemes for MPSoC architecture exploration given 4 to 8 processors in the MPEG2 decoder application

processor number, which indicates good scalability in estimating the MPSoC platform with different processor numbers.

To prove the feasibility of our work in architecture exploration, we obtain the total time consumption of the whole architecture exploration with different strategies, such as task mapping, processor types, and communication mechanisms given an MPSoC hardware platform. As shown in Fig. 14c, we run the automatic architecture exploration based on performance feedback in 2-, 3-, and 4-processor MPSoC hardware platforms. The experimental results show that the total time of VP is almost 15 833 s given a 2-processor MPSoC platform while it is increased to 38 527 s given a 4-processor MPSoC. Similarly, the other two schemes, TA and TA+VP, show that the total time is much increased when more processors are used. The result of the VA+TA scheme remains significant and even doubles if the processor number is increased, because the increasing processor number enlarges the design space of architecture exploration.

Another experiment on MPEG2 further shows the feasibility and efficiency of the VA+TA scheme with the proposed techniques. As shown in Fig. 14d, the optional MPSoC hardware platform has 4 to 8 processors because of the greater possibility of being parallelized in the MPEG2 function model. According to the complexity of the MPEG2 algorithm, the speed of the VP scheme is very slow, needing 53 h to explore the architecture given a 4-processor MPSoC platform. This is obviously an unacceptably long time. Moreover, it becomes worse when more processors are used, up to almost 7 d given an 8-processor MPSoC platform. We also see that the speed of the TA scheme is still too slow, about 3.2 h for an 8-processor MPSoC, which is also not fit to architecture exploration if more exploration strategies are used. However, the experiment of VA+TA shows a much better result on the total time consumed by architecture exploration, less than 75 min given a hardware platform with 4 to 8 processors. Furthermore, compared with the M-JPEG case, it is found that the total time curve of the VA+TA scheme keeps a similar changing trend in the MPEG2 experiment, which means that this scheme works efficiently with the increasing complexity of the functional model.

6.2.2 Estimation accuracy

Estimation accuracy is another key point in efficient architecture exploration. In this experiment we use the estimated total number of execution cycles

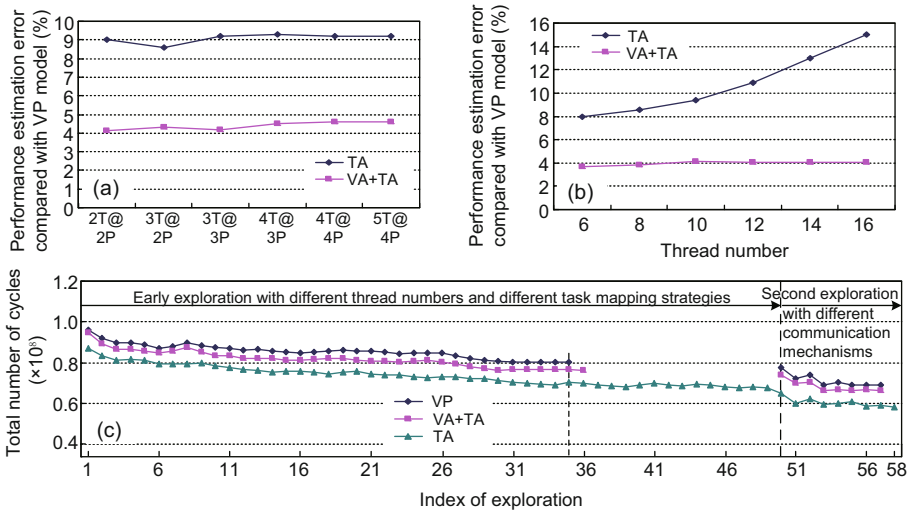

Fig. 15 Experimental results on estimation accuracy: (a) estimation error of TA and VA+TA with different numbers of processors and threads for the M-JPEG decoder application; (b) estimation error of TA and VA+TA given a 6-processor platform with 6 to 16 threads for the MPEG2 decoder application; (c) MPEG2 decoder architecture exploration on the 6-processor platform taking advantage of VP, VA+TA, and TA, respectively

of decoding a 10-frame M-JPEG or MPEG2 stream as the performance result. The strategy of VP takes advantage of the cycle-accurate VP model to obtain exact performance results in a low-level simulation. Therefore, we consider it as the reference to check the accuracy error of other strategies. Fig. 15a shows a comparison of the M-JPEG decoder performance estimation accuracy error between TA and VA+TA with different processor or thread numbers. Both strategies provide an estimation accuracy error of less than 10%. The strategy of VA+TA gains a better accuracy error at 4%–5%, compared with around 9% of the strategy of TA. The error grows a little with the increasing processor number because of the communication contention error caused by more memory access in parallel during TA simulation. The estimation error of the computation load of different processors may worsen the error of communication contention. In another experiment of MPEG2 decoding (Fig. 15b), the accuracy error is around 4% for the strategy of VA+TA, while the error of the strategy of TA grows rapidly with the increasing thread number. The strategy of VA provides almost the same error even as the application function becomes more complicated. Furthermore, its error is less sensitive to thread partition and mapping due to the fact that both dynamic and static factors are considered during profiling-based simulation and analysis. However, the strategy of TA uses only trace-based simulation and is not fit to the situation if thread partition and mapping are changed. As the experimental results show, the error of 16-thread estimation is almost twice that of 6-thread estimation, which becomes unacceptable if the number of threads keeps increasing.

The experimental results shown in Fig. 15c give the estimated performance point of each architecture during MPEG2 decoder architecture exploration on a 6-processor platform taking advantage of the strategies VP, TA, and VA+TA, respectively. The whole exploration process consists of two stages: the first is to use different partition and mapping strategies with 6 to 16 threads for early exploration of computation optimization, and the second is to explore different communication mechanisms for communication optimization. We see that the process of the first exploration stage is similar between VP and VA+TA. There are in total 35 architectures explored during architecture exploration using the strategy of VP for performance estimation. This number is increased to 36 using the strategy of VA+TA because of the performance estimation error. As for the final architecture of the first exploration stage, VP and VA+TA obtain the same result as a 12-thread partition and the same mapping. However, the strategy of TA provides a quite different process of the first

exploration stage with a total of 49 architectures explored. Its final architecture of this stage is a 16-thread partition, far from the result obtained by the strategy of VP, which indicates that the strategy of TA cannot be effectively used in architecture exploration. In the second stage of communication optimization, we see that the architecture exploration result curves of three strategies show a similar change because the same communication model is used in these three strategies. After the second exploration stage, we can obtain the same communication architecture using the strategies of VA+TA and VP. However, even given the similar change in the exploration curve, the strategy of TA still gives a different communication architecture result with more hardware FIFO channels. Thereby, we can see that the strategy of VA+TA is able to estimate the architecture accurately while maintaining a fast estimation speed, which very much improves the efficiency of architecture exploration.

7 Conclusions

In this paper, we present a profiling and annotation combined MPSoC performance estimation methodology and workflow from the VA level to the TA level. At the VA level, the accurate computation cost is obtained and annotated to the TA level. At the TA level, the communication latency is refined using an annotation based simulation method, which makes it more efficient for performance estimation. A series of experiments show the feasibility and efficiency of the proposed profiling and annotation combined performance estimation framework.

In future work, we will extend the performance estimation flow to the Simulink CAAM level analysis. The computation cost obtained at the VA level, the communication latency obtained at the TA level, and the coverage statistics will be analyzed and fed back to the Simulink model. By using the annotated Simulink model, the designer can refine the architecture in a more intuitive way.

References

ARM, 2003. AMBA Axi Protocol Specification v1.0.

Benini, L., Bertozzi, D., Bogliolo, A., *et al.*, 2005. MPARM: exploring the multi-processor SoC design space with SystemC. *J. VLSI Signal Process. Syst. Signal Image Video Technol.*, **41**(2):169-182. [doi:10.1007/s11265-005-6648-1]

Cesário, W.O., Nicolescu, G., Gauthier, L., *et al.*, 2001. Colif: a design representation for application-specific multiprocessor SoCs. *IEEE Des. Test Comput.*, **18**(5):8-20. [doi:10.1109/54.953268]

C-SKY Microsystems, 2013. Ck803 Introduction. Available from http://www.c-sky.com.

Filho, S.J., Aguiar, A., Marcon, C.A., *et al.*, 2008. High-level estimation of execution time and energy consumption for fast homogeneous MPSoCs prototyping. 19th IEEE/IFIP Int. Symp. on Rapid System Prototyping, p.27-33. [doi:10.1109/RSP.2008.25]

Fummi, F., Martini, S., Perbellini, G., *et al.*, 2004. Native ISS-SystemC integration for the co-simulation of multi-processor SoC. Proc. Design, Automation and Test in Europe Conf. and Exhibition, p.564-569. [doi:10.1109/DATE.2004.1268905]

Gao, L., Karuri, K., Kraemer, S., *et al.*, 2008. Multiprocessor performance estimation using hybrid simulation. Proc. 45th Annual Design Automation Conf., p.325-330. [doi:10.1145/1391469.1391552]

Gerin, P., Guerin, X., Pétrot, F., 2008. Efficient implementation of native software simulation for MPSoC. Proc. Design, Automation and Test in Europe, p.676-681. [doi:10.1109/DATE.2008.4484756]

Gerin, P., Hamayun, M.M., Pétrot, F., 2009. Native MPSoC co-simulation environment for software performance estimation. Proc. 7th IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis, p.403-412. [doi:10.1145/1629435.1629490]

GNU, 2013. gcov—a Test Coverage Program. Available from http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

Han, S.I., Baghdadi, A., Bonaciu, M., *et al.*, 2004. An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory. Proc. 41st Annual Design Automation Conf., p.250-255. [doi:10.1145/996566.996636]

Han, S.I., Chae, S.I., Jarraya, A.A., 2006. Functional modeling techniques for efficient SW code generation of video codec applications. Proc. Asia and South Pacific Design Automation Conf., p.935-940. [doi:10.1109/ASPDAC.2006.1594806]

Han, S.I., Chae, S.I., Brisolara, L., *et al.*, 2009. Simulink-based heterogeneous multiprocessor SoC design flow for mixed hardware/software refinement and simulation. *Integr. VLSI J.*, **42**(2):227-245. [doi:10.1016/j.vlsi.2008.08.003]

Henia, R., Hamann, A., Jersak, M., *et al.*, 2005. System level performance analysis—the SymTA/S approach. *IEE Proc.-Comput. Dig. Tech.*, **152**(2):148-166. [doi:10.1049/ip-cdt:20045088]

Huang, K., Han, S.I., Popovici, K., *et al.*, 2007. Simulink-based MPSoC design flow: case study of Motion-JPEG and H.264. Proc. 44th Annual Conf. on Design Automation, p.39-42. [doi:10.1145/1278480.1278491]

Huang, K., Yan, X.L., Han, S.I., *et al.*, 2009. Gradual refinement for application-specific MPSoC design from Simulink model to RTL implementation. *J. Zhejiang Univ.-Sci. A*, **10**(2):151-164. [doi:10.1631/jzus.A0820085]

Huang, K., Haid, W., Bacivarov, I., *et al.*, 2012. Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications.

ACM Trans. Embed. Comput. Syst., **11**(1), Article 8. [doi:10.1145/2146417.2146425]

Jerraya, A., Wolf, W., 2004. Multiprocessor Systems-on-Chips. Elsevier.

Jerraya, A.A., Bouchhima, A., Petrot, F., 2006. Programming models and HW-SW interfaces abstraction for multi-processor SoC. 43rd ACM/IEEE Design Automation Conf., p.280-285. [doi:10.1109/DAC.2006.229246]

Karuri, K., Al Faruque, M.A., Kraemer, S., *et al.*, 2005. Fine-grained application source code profiling for ASIP design. Proc. 42nd Design Automation Conf., p.329-334. [doi:10.1109/DAC.2005.193827]

Keutzer, K., Newton, A.R., Rabaey, J.M., *et al.*, 2000. System-level design: orthogonalization of concerns and platform-based design. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **19**(12):1523-1543. [doi:10.1109/43.898830]

Kienhuis, B., Deprettere, E., Vissers, K., *et al.*, 1997. An approach for quantitative analysis of application-specific dataflow architectures. Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors, p.338-349. [doi:10.1109/ASAP.1997.606839]

Kirchsteiger, C.M., Schweitzer, H., Trummer, C., *et al.*, 2008. A software performance simulation methodology for rapid system architecture exploration. 15th IEEE Int. Conf. on Electronics, Circuits and Systems, p.494-497. [doi:10.1109/ICECS.2008.4674898]

Madl, G., Dutt, N., Abdelwahed, S., 2007. Performance estimation of distributed real-time embedded systems by discrete event simulations. Proc. 7th ACM & IEEE Int. Conf. on Embedded Software, p.183-192. [doi:10.1145/1289927.1289958]

Oyamada, M., Wagner, F.R., Bonaciu, M., *et al.*, 2007. Software performance estimation in MPSoC design. Proc. Asia and South Pacific Design Automation Conf., p.38-43. [doi:10.1109/ASPDAC.2007.357789]

Oyamada, M., Zschornack, F., Wagner, F., 2008. Applying neural networks to performance estimation of embedded software. *J. Syst. Archit.*, **54**(1-2):224-240. [doi:10.1016/j.sysarc.2007.06.005]

Patel, R., Rajawat, A., 2011. A survey of embedded software profiling methodologies. *Int. J. Embed. Syst. Appl.*, **1**(2):19-40. [doi:10.5121/ijesa.2011.1203]

Piscitelli, R., Pimentel, A.D., 2012. Interleaving methods for hybrid system-level MPSoC design space exploration. Int. Conf. on Embedded Computer Systems, p.7-14. [doi:10.1109/SAMOS.2012.6404152]

Posadas, H., Herrera, F., Sanchez, P., *et al.*, 2004. System-level performance analysis in SystemC. Proc. Design, Automation and Test in Europe Conf. and Exhibition, **1**:378-383. [doi:10.1109/DATE.2004.1268876]

Richter, K., Jersak, M., Ernst, R., 2003. A formal approach to MPSoC performance verification. *Computer*, **36**(4):60-67. [doi:10.1109/MC.2003.1193230]

Schnerr, J., Bringmann, O., Viehl, A., *et al.*, 2008. High-performance timing simulation of embedded software. Proc. 45th Annual Design Automation Conf., p.290-295. [doi:10.1145/1391469.1391543]

Shen, H., Hamayun, M., Petrot, F., 2012. Native simulation of MPSoC using hardware-assisted virtualization. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **31**(7):1074-1087. [doi:10.1109/TCAD.2012.2187526]

Wandeler, E., Thiele, L., Verhoef, M., *et al.*, 2006. System architecture evaluation using modular performance analysis: a case study. *Int. J. Softw. Tools Technol. Transfer*, **8**(6):649-667. [doi:10.1007/s10009-006-0019-5]

Wilhelm, R., Engblom, J., Ermedahl, A., *et al.*, 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, **7**(3):36. [doi:10.1145/1347375.1347389]

Yang, H., Kim, S., Ha, S., 2010. An MILP-based performance analysis technique for non-preemptive multitasking MPSoC. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, **29**(10):1600-1613. [doi:10.1109/TCAD.2010.2061552]